On AC^0 Implementations of Fusion Trees and Atomic Heaps

Mikkel Thorup*

Abstract

Addressing a question of Fredman and Willard from STOC'90, we show that fusion trees cannot be implemented using the AC^0 operations available through a standard programming language such as C. However, they can be implemented using AC^0 operations on emerging multimedia processors such as the Pentium 4.

A fusion node is a linear space representation of an integer set X of size $O(\sqrt{W})$, where $W \geq \log n$ is the word-length. The fusion node supports searches in X in constant time. Here, a search for y in X returns $\max\{x \in X | x \leq y\}$. Using fusion nodes in a $O(\sqrt{W})$-degree fusion tree gave Fredman and Willard $O(\log n / \log W)$ searching for general n, beating the comparison based lower-bound. However, the search routine uses multiplication which is not an AC^0 operation.

Fredman and Willard asked if multiplication instructions could be avoided. We show that the answer is "no" unless you have room for a multiplication table. More precisely, restricting ourselves to the AC^0 operations available through C, we show that constant time look-ups or searches in sets of *any* non-constant size require space $2^{\Omega(W)}$. However, if we have that much space, i.e., $2^{\varepsilon W}$ for some constant $\varepsilon > 0$, then we can tabulate multiplication of $(\varepsilon W/2)$-bit numbers, and then we get constant time multiplication of words using additions and shifts. Previous related lower-bounds all disallowed some common AC^0 instructions in C such as shifts.

We note that even on the weaker "Practical RAM" the above $2^{\Omega(W)}$ space lower-bound for constant look-ups was only known for sets of size $\Omega(W^2)$ (Miltersen, ICALP'96). Our $\omega(1)$ set size is best possible since sets of constant size can be searched directly in constant time.

Contrasting the above result, we show that using the AC^0 operations available on Intel's new Pentium 4, we can implement both fusion trees and Fredman and Willard's later atomic heaps from FOCS'90. Among the many consequences, we get linear time and space AC^0 implementations of minimum spanning tree and undirected single source shortest paths. Also, we get optimal $\Theta(\log n / \log \log n)$ implementations of dynamic rank and $1\frac{1}{2}$ dimensional range searching. Previous optimal solutions required either multiplication or the use of self-designed AC^0 instructions not available on existing processors.

1 Introduction

In 1990 Fredman and Willard [21] introduced fusion trees allowing them to sort integers in $O(n \log n / \log \log n)$ time, thus breaking the $\Omega(n \log n)$ lower bound for comparison based algorithms. The model used is what we can program on a real computer, using the fact that integers cannot only be compared but also added and multiplied, and be used for addressing. The time bound assumes that the integers fit in words so that they can be operated on in constant time; otherwise even comparisons take more than constant time. Later the same year [22], using the same basic ideas, Fredman and Willard introduced atomic heaps, allowing them to find minimum spanning trees in a graph with integer weights in linear time and space. Since then these fundamental data structures have found many other applications (see e.g. [4, 6, 36, 33]). As pointed out by Fredman and Willard, a theoretical objection to fusion trees and atomic heaps is that they use multiplication which is not an AC^0 instruction. As their final question, describing "brisk" as AC^0 and "RISC" as normal machine instructions, they asked "Can fusion trees be implemented with Brisk RISC?"

Addressing this question, we show that fusion trees and atomic heaps cannot be implemented with the AC^0 instructions available in a standard programming language such as C [26].

Contrasting this result, we show that fusion trees and atomic heaps can be implemented using the AC^0 instructions available on Intel's new Pentium 4 [25], thus providing linear time and space AC^0 implementations of minimum spanning tree and single source shortest paths on a Pentium 4.

Fusion trees and atomic heaps The basic idea of fusion trees is that we can search a sufficiently small integer set S in constant time. Here, by *searching x in S*, we mean finding the largest integer in S which is at most as large as x. This follows the general framework of Gabow and Tarjan [23] of providing constant time solutions for small problems, and apply these as subroutines on subproblems of a larger problem.

More concretely, let $W \geq \log n$ be the word-length. The essence of Fredman and Willard's [21] *fusion trees* is that given a set S of $d \leq \sqrt[6]{W}$ in $O(d^4)$ time, we can construct an $O(d)$ space data structure supporting searches in S in constant time. Given a set X of n integers, Fredman and Willard used these data structure

*AT&T Labs—Research, 180 Park Avenue, Florham Park, NJ 07932, USA. Fax: +1 (973) 360 8178, Phone: +1 (973) 360 8904, E-mail: mthorup@research.att.com

to implement a B-tree of degree $O(\sqrt[6]{W})$ over the n keys with a height and search time of $O(\log n/\log W + 1) = O(\log n/\log\log n)$. They also used this to get deterministic linear space sorting in $O(n\log n/\log\log n)$ time.

For $n = W^{O(1)}$ keys, the B-trees give us linear space searching in constant time. It follows from the results of Beame and Fich [12] that sets of larger size, i.e., of size $W^{\omega(1)}$, cannot be represented in polynomial space with constant search time.

Fredman and Willard's [22] later *atomic heaps* have a similar flavor, but more dynamic: using $O(n)$ space and preprocessing time, they maintain a family of dynamic sets, each of size $O(\sqrt{\log n})$, supporting insert, delete, search and rank in $O(1)$ time. Here, by *rank* of x in S we mean the number of elements in S that are no bigger than S. As a prime application, Fredman and Willard show how to implement minimum spanning tree with integer weights in linear time and space.

Note that atomic heaps do not replace fusion trees for it may be that W is much larger than $\log n$. For example, Fredman and Willard [21] combine their $O(\log n/\log W + 1)$ bound with the $O(\log W)$ bound of van Emde Boas [35] to get searching in $O(\sqrt{\log n})$ time. However, to get linear, or just polynomial, space with van Emde Boas's data structure, we need multiplication based hashing [28].

In the applications of atomic heaps in deterministic linear time and space algorithms for minimum spanning trees [22] and undirected single source shortest paths [33], it would actually suffice with sets of size

$$\overbrace{\log\log\cdots\log}^{i} n \tag{1.1}$$

for any constant i. Thus, our lower-bounds should work for that small sets.

Instruction sets First we note that the basic fusion tree idea of searching a sufficiently small set in constant time was really originated in 1983 by Ajtai et al. [2], but using specialized instructions not available on any real processor. The main contribution of Fredman and Willard [21] was to show that such ideas could be implemented using normal machine instructions, with a very clever use of multiplication.

However, multiplication is not an AC^0 operation, that is, there are no circuits for multiplication of constant depth and of size (number of gates) polynomial in the word length. Here, the gates are negations, and $\wedge$- and $\vee$-gates with unbounded fan-in. More precisely, the minimal depth of a polynomially sized circuit for multiplication is $\Theta(\log W/\log\log W)$ [13]. The depth with polynomial size is a model of the time it takes to execute an instruction, so with the standard assumption that $W = \Omega(\log n) = \omega(1)$, AC^0 is the class of functions which can be computed in constant time in this model. Fredman and Willard raised as an open problem if fusion trees could be implemented with AC^0 operations.

The above problem was addressed in 1999 by Andersson et al. [9] who showed that using specialized non-standard AC^0 operations, we can implement fusion trees. They also pointed out that the same technique works for atomic heaps. However, this still leaves open what can be done with standard AC^0 operations.

Standard AC^0 operations In our view, *standard operations* are those available via a standard programming language such as C [26]. We have a processor determined word-size W, limiting how big integers we can operate on in constant time. We assume that each input integers fits in a single word.

By *standard AC^0 operations*, we mean standard operations where all the computational instructions are in AC^0. Here, by computation instructions, we mean operations computing a word from a constant number of other words. In AC^0, this includes addition, subtraction, comparisons, shifts, logical and bit-wise boolean operations, and conversions between representations, the non-trivial ones being between integers and floating point numbers. AC^0 basically rules out multiplication and division. Not affected by the AC^0 constraint are non-computational instructions such as assignment statements with direct and indirect addressing, and (computed) `goto`s. As our main result, we prove

THEOREM 1.1. *For any $b < \sqrt{W}$, $b = \omega(1)$, there is a concrete set X of size 2^b so that any representation of X using less than $2^{W/b}$ space spends at least $\Omega(\log b)$ time on some look-ups if restricted to standard AC^0 operations.*

Above, a *look-up* of a value x in X returns a pointer to an element in X with value x, if any, and this pointer can be used to look up satellite information stored with x. Clearly look-ups are no harder than searches, for we can always append pointers to stored values and then use searches to find an element including its appended pointer.

COROLLARY 1.1. *With standard AC^0 operations, constant time look-ups or searches in worst-case sets of any non-constant size require space $2^{\Omega(W)}$. In fact, for any constant $\varepsilon > 0$, space $2^{\varepsilon W}$ suffices for sets of size $W^{O(1)}$.*

Proof. For the lower-bound, consider $n = \omega(1)$ and $\alpha = o(1)$. We want to construct a set of size n so that any representation of size $2^{W\alpha}$ requires non-constant look-up time in the worst case. We apply Theorem 1.1

with

$$b = \lfloor \min\{\log n, 1/\alpha\} \rfloor = \omega(1).$$

The resulting set X_b has size at most n. We augment X_b arbitrarily to a set X of size n. Any representation of look-ups in X also provides look-ups in X_b. Hence a representation of size at most $2^{W\alpha} \leq 2^{W/b}$ spends $\Omega(\log b) = \omega(1)$ time on some look-ups.

For the upper-bound, we just note that we can tabulate multiplication of $W/(2\varepsilon)$-bit numbers, and this allows us to multiply in constant time, and hence we can implement Fredman and Willard's fusion trees.

This result is in sharp contrast to the fusion trees that give linear space representations of sets of size $W^{O(1)}$, supporting searches and look-ups in constant time, using either multiplication [21] or specialized self-defined AC^0 operations [9]. As mentioned under atomic heaps, it is important that our lower-bounds cover sets as small as those defined by (1.1).

We note that related lower-bounds are known with more restricted instruction sets. For example, Ben-Amram [14] have shown that for a computer whose computational instructions are arithmetic and boolean operations, we need expensive double-precision multiplication to sort in $o(n \log n)$ time. However, their model exclude standard shifts, and including shifts, Andersson et al. [7] have shown that we can sort in $O(n \log\log n)$ time. In the same model, excluding double-precision multiplication, Fich and Miltersen have shown that membership queries take $\Theta(\log n)$ time unless we have a representation using $2^W/n^{o(1)}$ space. Again, this lower-bound is violated by shifts, with which we get constant time using linear space with two-level hashing of Fredman et al. [19].

Miltersen [29] have shown that on the so-called "Practical RAM" with computational instructions are limited to additions, shifts, and bit-wise boolean operations, sets of size at least W^2 require $2^{\Omega(W)}$ space for constant look-ups.

Expanding on Miltersen's technique, Brodnik et al. [16] have shown that if the computational instructions are limited to additions, shifts, and bit-wise boolean operations, then the msb operation of computing the position of the most significant set bit in a word takes $\Omega(\log\log W)$ time with space $O(2^{W^{1-\varepsilon}})$ for any fixed $\varepsilon > 0$. This immediately gives a non-constant lower-bound for searching and member-ship, for we can simply consider the W words with a single set bit.

However, a lower-bound is only relevant for fusion trees if it works for sets of size W^ε for any ε. The standard AC^0 operations subsumes those of the Practical RAM, so Theorem 1.1 and Corollary 1.1 improves all the above Practical RAM results from the perspective of dictionaries.

More importantly, a major objection to the pure Practical RAM lower-bounds is that msb, which is proved non-constant time on the Practical RAM, is, in fact, computed in constant time with standard AC^0 operations. As noted by Andersson and Thorup [10], the msb of a x can be obtained by converting x to a floating point numbers and then extracting the exponent (c.f. [24]). Conversion to floating point representation is a very commonly used AC^0 operation, and the exponent can be extracted with a standard shift. In fact, this msb implementation was found to be very fast in experiments.

Since msb is one of the key operations coded by multiplication in fusion trees, it is clearly not reasonable to exclude it when proving lower-bounds for fusion trees.

The Pentium 4 The above lower-bound says that we cannot expect to find a portable AC^0 implementation of fusion trees. However, this does not rule out the possibility of finding an AC^0 implementation of fusion trees on a concrete processor. We show here that indeed, they can be implemented in AC^0 on Intel's Pentium 4 [25], which is one of the recent multimedia processors. We note that given the Pentium 4, the implementation is in itself not so difficult. The main effort going into this paper has mostly been, over 3 years, to keep a patient eye on new multimedia processors, waiting for one with a sufficiently powerful instruction set (it is graphics and not fusion trees that push processor design, so my own efforts to convince hard-ware people that they needed these instructions were fruitless).

This gives the first optimal AC^0 solutions for many of the problems currently solved using fusion trees or atomic heaps. A sample of such fundamental problems is given below. For all of them, we assume that the weights or values are integers stored in words.

- Minimum spanning tree in deterministic linear time and space using Fredman and Willard's algorithm [22]. The previous best deterministic AC^0 solution is the comparison based algorithm of Chazelle using $O(m\alpha(m,n))$ time [17].
- Undirected single source shortest paths tree in deterministic linear time and space using Thorup's algorithm [33]. The previous best deterministic AC^0 solution takes $O(m\alpha(m,n))$ time [33].
- Dynamic rank in linear space supporting insert, delete, and rank in $O(\log n/\log\log n)$ time. As pointed out by Andersson et al. [9], we just combine Fredman and Willard's $O(\log n/\log\log n)$ searching [21, 36] with Dietz' dynamic lists [18]. As shown by Fredman and Saks [20], this time bound is optimal even with an arbitrary non-AC^0 instruction

set. The previous best AC^0 solution is the standard comparison based $O(\log n)$ solution.

- Dynamic priority trees supporting insert, delete, and $1\frac{1}{2}$ dimensional range queries in $O(\log n/\log\log n)$ time, using the algorithm of Willard [36]. As shown by Alstrup et al. [5], this time bound is optimal even with an arbitrary non-AC^0 instruction set. The previous best AC^0 solution is the original $O(\log n)$ solution by McCreight [27].

The reader is referred to Willard's paper [36] for many other applications of atomic heaps that can now be implemented with AC^0 operations on a Pentium 4.

It is interesting to note that our result has no impact on the original application of fusion trees: sorting and searching. For sorting, we already have a deterministic $O(n(\log\log n)^2)$ time and linear space solution using only standard AC^0 operations [32]. For searching, fusion trees (not atomic heaps) are still used in the optimal $\Theta(\sqrt{\log n/\log\log n})$ time static and dynamic solutions [6, 12, 11]. However, these results also involve hashing which, as proved by Andersson et al. [8], is impossible to implement efficiently with AC^0 operations, even if we are allowed arbitrary non-standard AC^0 operations.

Implementability matters to theory In response to earlier criticism of this paper, we here stress that implementability is a very important aspect of computational models. Using their own specialized instructions, Ajtai et al. [2] had essentially implemented the fusion trees, or rather an atomic heap, 7 years before Fredman and Willard presented their famous fusion trees [21]. A mathematician might just think of fusion trees as a reimplementation of the result of Ajtai et al. in yet another computational model, but the result was very exiting to theoretical *computer* science, exactly because it only used normal machine instructions to encode the specialized instructions of Ajtai et al. The encoding was based on a clever unconventional use of multiplication. Of course, the implementability on a normal machine does imply practicality. Fredman and Willard followed the theoretical tradition of only investigating asymptotic running times, and indeed the constants hidden in the O-notation were way too large for their result to be of practical relevance.

There are plenty of results like the one of Ajtai et al. [2] stating that we can do interesting stuff if we get to design our own specialized computer. For example, Fredman and Saks [20] suggested RAMBO (random access with byte overlap) and Brodnik et al. [15] have shown that with RAMBO, we can do searching in constant time and linear space, breaking the $\Omega(\sqrt{\log n/\log\log n})$ lower-bound of Beame and Fich [12] for searching in standard memory using polynomial space. Clearly, it is very important that theory investigates the power of alternative computational features, but that does not in general imply that RAMBO algorithms are as important as algorithms that can actually be implemented in C to run on almost any real computer. Our lower-bound in Corollary 1.1 shows that no C algorithm can exist which codes anything like fusion trees or atomic using only AC^0 operations.

Our investigation of the Pentium 4 takes the implementability one step down. Having proved that we cannot do the AC^0 implementation on todays computer in C, we try to look into the multimedia processors of tomorrow. This follows the spirit of PRAM algorithms. These received a lot of attention while people believed that they modeled important computational devises of the future. PRAMs are still equally beautiful from a theoretical viewpoint, but the interest vanished because industry eventually gave up producing efficient PRAM-like computers. Our work on multimedia processors is a lot more conservative, in that we stick to instructions that have already been successfully implemented on some popular off-the-shelf processors.

2 Lower-bound with standard AC^0 operations

In this section, we will show the lower-bound of Theorem 1.1. Our proof will use the framework developed for by Brodnik, Miltersen, and Munro [29, 16] for proving lower-bounds on the Practical RAM.

Informally speaking, on the Practical RAM they showed that for a quick query routine running in close to constant time, unless we have a huge table, any small segment of output bits can be fixed for most inputs by fixing of a corresponding set of input bits, no matter how the fixing is done.

This kind of statement is false for msb, for if we consider the $\log W$ least significant bit, then, if we fix any $O(\log W)$ bits to 0, we still have $W - O(\log W)$ possible outcomes in our output segment. Using this, Brodnik et al. [16] get a lower-bound for msb, and hence for look-ups in the set of words with a single 1.

However, as mentioned in the introduction, msb can be implemented in constant time in C using conversion to floating point numbers and shifts, all of which are AC^0 operations.

Adding msb to the Practical RAM clearly breaks down the above informal characterization of quick queries. We therefore have to develop a new more complex characterization to of quick queries that admits msb-operations. In the process, we actually end up with improved bounds, even for the Practical RAM. More precisely, we get non-constant look-ups for sets of any

non-constant size whereas [29, 16] needed the sets to be of size W, which is too big to rule out fusion trees.

We now define a concrete *standard AC^0 RAM* which is simple for proving lower-bound, yet sufficiently powerful to run C programs with standard AC^0 opertions incurring only a constant factor slow-down in time.

The set of all W bit words is denoted $\mathcal{W}$. Our processor has a set $\mathcal{R}$ of three registers A, B, and C, each containing a word, and a random access memory $\mathcal{M}$, which is viewed as an array of words. We have the following statements available: X $\leftarrow$ Y copying the contents of Y $\in \mathcal{R}$ to X $\in \mathcal{R}$; X $\leftarrow c$ assigning X $\in \mathcal{R}$ a constant $c \in \mathcal{W}$; A $\leftarrow \mathcal{M}$[B] reading the memory word specified by B into A; $\mathcal{M}$[B] $\leftarrow$ A writing the contents of A into the memory word specified by B; **if** A $= 0$ **goto** B conditional jump to the program location specified by register B if A is 0; A $\leftarrow$ A $\ll$ B shifting A to the left by the amount specified by B (a right shift if B is negative); A $\leftarrow$ A$\bar{\wedge}$B nanding A and B bitwise in A (this suffices to simulate all bit-wise boolean operations in constant time); A $\leftarrow$ A+B adding A and B in A (we can implement subtraction using $-x = \neg x + 1$ where $\neg x = x\bar{\wedge}x$ is bitwise negation); A $\leftarrow$ msb(a) assigning A the position of its most significant bit, the least significant bit being at position 0 (if A $= 0$, the value is 0). Here that the less significant bits are to the right. When the program start, all three registers a 0, so the first statements will typically assign them some constants.

We note that it is really standard to compile multiplication/division-free C programs onto a processor like the one above [1], using msb to implement floating point instructions.

Concerning resources, time is simply the number of statements performed. The space is the maximum of the maximum memory location and the maximum program location considered. We need here to include the program in the space; for otherwise, we could code a gigantic table by jumping to program locations assigning appropriate constants to the registers. Our lower-bounds actually admits self-modifying code though we will ignore this complication below.

As an aid in showing our lower bounds, we shall consider *word-circuits* computing on words rather than just on Booleans. Each wire of such a circuit C holds a W-bit word. It may contain four kinds of gates: +-gates, $\bar{\wedge}$-gates, msb-gates, and $\ll_r$-gates shifting left with some constant value r that if negative indicates a right shift. Arbitrary word constants c may also be fed into the circuit. We note that having the shift amount a constant is an idea of Miltersen [29] which is very important for the analysis below. The size of a circuit is the number of gates in contains. A circuit with one source and one sink computes a function from words to words.

We will need a generic hard set $HARD_b$, which for $b \leq W$ denotes the set of 2^b words that only vary in bit-positions $i\lceil W/b \rceil$, $i = 0, ..., b-1$. One of the ideas behind this set is that relative to its size, it minimizes the number of outcomes for msb.

LEMMA 2.1. *Let $2 \leq b \leq W$. Let C be a word circuit of size s. Let u be a word with a single block of ones in the least significant end, that is, $u = 0^{W-k}1^k$. Then there is a bit string v_C with the following properties*

(i) *v_C has length $2W$ and will be indexed $v_C[-W]\cdots v_C[-1]v_C[0]v_C[1]\cdots v_C[W-1]$, so when we do bit-wise $\wedge$ between a word and v_C shifted, only the part of v_C aligned with the word will matter.*

(ii) *v_C has at most 2^s segments of 1s, each of length at most k.*

(iii) *for any $i = 0, ..., W-1$, if $x \in HARD_b$ varies subject to $x \wedge (v_C \ll i)$ being fixed, i.e., x may only vary in bits where $v_C \ll i$ is 0, then $C(x) \wedge (u \ll i)$ takes on at most $2^{2^{2s+\log\log b}}$ values.*

In our later applications, we will have 2^s and k much smaller than b and W/b, respectively, so v will be far from fixing all the variable bits in $HARD_b$. The corresponding Lemma 14 in [16] does not restrict the inputs to be in a certain set. Also, they fix $2^s k$ arbitrary bits, rather than 2^s segments of k consecutive bits. The more interesting changes, however, are in the proof.

Proof. For a given circuit C, we construct v_C by induction of the size s of C. We define t_C to be the number of segments from (ii). These are allowed to touch. We define p_C to be the maximum number of values from (iii). If s is the size of C, we want $t_C \leq 2^s$ and $p_C \leq n_s = 2^{2^{2s+\log\log b}}$.

In order to deal with msb-gates, we consider two quantities q_C and r_C such that words in $C(HARD_b) = \{C(x) | x \in HARD_b\}$ can be divided into q_C segments, each taking at most r_C values. Thus, we may have $|C(HARD_b)| = r_C^{q_C}$. However, $|\text{msb}(C(HARD_b))| = r_C q_C$ since each of the q_C segments can provide at most r_C msb-values.

In one combined induction, we construct the v_C so that $t_C \leq 2^s$ and p_C, q_C, and r_C are bounded by n_s. For bounds in terms of n_s, we note that $n_0 = b \geq 2$ and $n_s = n_{s-1}^4 > 2n_{s-1}^2$.

If $s = 0$, $C(x) = x$ or $C(x) = c$ for some constant c.. In both cases, we pick $v_C = u$. Then $t_C = 1 = 2^0$. If $C(x) = c$, we trivially have $p_C = q_C = r_C = 1$. If $C(x) = x$, we also have $p_C = 1$ since we have fixed

the relevant bits. We divide the words in $C(HARD_b) = HARD_b$ into $q_C = b$ segments, each containing one of the varying bit positions in $HARD_b$. Then each segment takes only $r_C = 2$ values over $HARD_b$.

Now, consider C of size $s > 0$. Suppose $C = \mathsf{msb}(D)$. Then $v_C = v_D$ and $|C(X)| \leq q_D r_D \leq n_{s-1}^2$, and this bounds both p_C and q_C with $r_C = 1$.

Now, suppose $C = D \bar{\wedge} E$. Then $v_C = v_D \vee v_E$ and $t_C \leq t_D + t_E \leq 2 \cdot 2^{s-1} = 2^s$. We divide the output words into segments using all end-points of segments from the outputs of C and C. Then we get $q_C \leq q_D + q_E - 1 < 2n_{s-1} < n_s$ output segments. Each output segment combines values of one output segment from each of C and D, so $r_C \leq r_D r_E \leq n_{s-1}^2$. Similarly, for the segment determined by u, we get $p_C \leq p_D p_E \leq n_{s-1}^2$.

Finally, suppose $C = D + E$. We use the same division into segments as for the $\bar{\wedge}$. However, this time, we get twice as many values for the segments because we may, or may not, have a carry bit affecting our result. Thus, we get $r_C \leq 2 r_E r_D \leq 2n_{s-1}^2$ and $p_C \leq p_D p_E \leq 2n_{s-1}^2$.

Finally, if $C = \ll_r (D)$, we only have to shift v in the opposite direction. Then all the measures remain unchanged, that is, $p_C = p_D$, $q_C = q_D$, $r_C = r_D$.

LEMMA 2.2. *Let $2 \leq b \leq W$. Let C be a word circuit of size s. Suppose $C(x) < 2^{W/b}$ for all $x \in U \subseteq HARD_b$. Then there is a subset $V \subseteq U$ of size at least $|U|/2^{2^{2s+1+\log\log b}}$ such that $|C(V)| = 1$.*

Proof. We apply Lemma 2.1 with $k = \lceil W/b \rceil$ and $i = 0$, getting some v_C satisfying (i), (ii), and (ii). By definition, $C(x) \wedge u = C(x)$ for all $x \in U$. Each of the 2^s segment of 1s in v_C are too short to fix more than on of the variable input bits from $U \subseteq HARD_b$, so v_C can only fix 2^s input bits. Since there are only 2^{2^s} possible fixings of these bits, we can find a subset U' of U which is fixed according to v_C, and of size $|U'| \geq |U|/2^{2^s}$. On the other hand, since $C(x) \wedge u = C(x)$ for all $x \in U$, by (iii), $|C(U')| \leq 2^{2^{2s+\log\log b}}$. Hence, we can find a subset V of U' with $|C(v)| = 1$ of size $|V| \geq |U'|/2^{2^{2s+\log\log b}} \geq |U'|/2^{2^{2s+1+\log\log b}}$.

We are now ready the following more detailed version of our main result from Theorem 1.1.

THEOREM 2.1. *For $2 \leq b \leq \sqrt{W}$, let $HARD_b$ be the set of 2^b words from Lemma 2.1. Then any representation of $HARD_b$ using less than $2^{W/b}$ space spends $\log b/3$ time on look-ups on the standard AC^0 RAM.*

Proof. We assume $W = \omega(1)$. Consider executing some look-up algorithm L on inputs $x \in HARD_b$. We are going to develop word circuits to track the execution of the algorithm for a large subset of the input. Time steps between integers, starting at 0. For each time t, we will construct a subset $U_t \subseteq HARD_b$ of the inputs, memory locations $m_0, ..., m_{t-1}$ and word circuits $R_1^A, R_1^B, R_1^C, ..., R_t^A, R_t^B, R_t^C$ satisfying the following conditions:

- $|U_t| \geq u_t = 2^b/2^{2^{2t+\log\log b}}$.
- The size of each word circuit, R_i^X is at most i.
- For all inputs $x \in U_t$, at each time $i \leq t$, the program counter was at the same location and the value of register X = A, B, C was $R_i^X(x)$. Moreover, if m_i is defined, at time i, we had $R_i^B(x) = m_i$ and wrote $R_i^A(x)$ into memory location $\mathcal{M}[m_i]$. At time t, $\mathcal{M}[m]$ has its original value if $m \notin \{m_0, ..., m_{t-1}\}$. Otherwise, $\mathcal{M}[m] = R_i^A(x)$ where i is the largest index with $m_i = m$.

Let us first see that the conditions imply the theorem. Suppose our algorithm L completes the look-up in T steps. Without loss of generality, we can assume the output is an index in $\{0, ..., 2^b - 1\}$; for these indices can be stored with the keys, so if we can find the keys, we can also find the indices. In fact, we can even require this output to be in register A. Thus, for $x \in U_T$, the output is $R_T^A(x) < 2^b$. Since $b \leq W/b$, by Lemma 2.2, we can find a subset $V \subseteq U_T$ with $|R_T^A(V)| = 1$ of size $|U_T|/2^{2^{2T+1+\log\log b}} \geq 2^b/2^{2^{2(T+1)+\log\log b}}$. If $|V| > 1$, this implies that we have two input keys from $HARD_b$ producing the same look-up, thus demonstrating an error in the look-up algorithm. We therefore conclude that $T > \log b/3$, as stated in Theorem 2.1.

We should now show how to satisfy our conditions as we go from time t to time $t+1$. The computations on registers are all straightforward, e.g., if the instruction at time t is A ← A + B, we just set $R_{t+1}^A = R_t^A + R_t^B$, leaving the other circuits unchanged. If there is a write $\mathcal{M}$[B] ← A, we set $m_t = R_t^B$, noting that there is a space error if $m_t \geq 2^W/b$. All other instructions would leave m_t undefined.

The most interesting case is that of a read A = $\mathcal{M}$[B]. Unless we have a read error, for each $x \in U_t$, $R_t^B(x) < 2^{W/b}$, and hence, by Lemma 2.1, we can find a subset U_{t+1} of U_t with $|R_t^B(U_{t+1})|$ of size $|U_{t+1} \geq |U_t|/2^{2^{2t+1+\log\log b}} \geq 2^b/2^{2^{2(t+1)+\log\log b}}$.

Let m be the unique value in $R_t^B(U_{t+1})$. If $m \notin \{m_0, ..., m_{t-1}\}$, $\mathcal{M}[m]$ has its original value c, and then we set if $R_{t+1}^A = c$. Otherwise, let i be the largest index with $m_i = m$. Then $R_{t+1}^A = R_i^A$. All other circuits are left unchanged.

The conditional jump is very similar. First, let U' be the set of $x \in U_t$ with $R_t^B(x) > 0$. If this is more

than half, we simply set $U_{t+1} = U'$. Otherwise, we use that all program locations have to be below $2^{W/b}$ to find a subset $U_{t+1} \subseteq U_t \setminus U'$ of inputs that all have A = 0 and agree on the program location B that we jump to. The set is of size $(|U_t|/2)/2^{2^{2t+1+\log\log b}} \geq 2^b/2^{2^{2(t+1)+\log\log b}}$. None of the circuits have to be changed.

Finally, for the shifts, we exploit that the maximal shift amount is $\log W = o(W/b)$. Again this gives us a large enough subset $U_{t+1} \subseteq U_t$, agreeing on the shift amount r, and then we just set $R^A_{t+1} = \ll_r (R^A_{t+1})$, leaving all the other circuits unchanged.

3 Multimedia processors

A multimedia processor like Intel's Pentium 4 [25] offers many instructions for dealing with 64- and 128-bit registers as vectors of 8-, 16-, 32-, and/or 64-bit fields. With the field length understood, let $X\langle i\rangle$ denote field i of X, with $X\langle 0\rangle$ denoting the right most and least significant field. Similarly, we let $X[i]$ denote bit i of X, with $X[0]$ denoting the right most and least significant bit.

A small curiosity is that a field representing a Boolean field comparisons such as = and < return all 1s if true and all 0s if false. For example, we have $\texttt{VecNEQ}(X,Y)$, setting $X\langle i\rangle$ to all 1s if $X\langle i\rangle \neq Y\langle i\rangle$, and to all 0s otherwise [25, pp. 3-348–551].

Notationally, we will use generally use the prefix Vec- to indicate a vector operation working coordinate wise on fields. Vectors are always upper case letters whereas regular integers are lower-case.

We note that the view of words as vectors of fields fits very well with the way words are treated in many bit-fiddling sorting algorithms. For example, in [3, 34], Albers and Hagerup and Thorup use a routine $\texttt{Mask}(X)$ setting $X\langle i\rangle$ to all 1s if $X\langle i\rangle$ contains a 1. Subsequently, X is used to select fields in other vectors. Implementing Mask with standard AC^0 operations over full words requires some care, but with VecNEQ, we simply set $\texttt{Mask}(X) = \texttt{VecNEQ}(X, 0)$.

In addition to coordinate-wise vector operations on fields, we have some operations for communication between coordinates. For our purposes, the following two operations are particularly important:

- $y = \texttt{PackSignBits}(X)$ [25, p. 3-576]: Returns the most significant bit in each field, that is, $y[i] = X\langle i\rangle[0]$. This operation is trivially an AC^0 operation.
- $Y = \texttt{Shuffle}(X, Z)$ [25, p. 3-605–611]: The fields in Z are viewed as field pointers; a field is returned, containing the selected fields in X. That is, $Y\langle i\rangle = X\langle Z\langle i\rangle\rangle$.

  To see that Shuffle is an AC^0 operation, we note that for $K \times L$ bit vectors, the output bits are determined by

$$\forall i \leq K, j \leq L : Y[iL + j] = \bigvee_{r=0}^{K-1} (r = Z\langle i\rangle \wedge X[rL + j]),$$

  corresponding to a naive depth 2 circuit of size $O(KL^2)$.

THEOREM 3.1. *Let $K = 2^k$, $L = 2^\ell$, and $M = 2^m \geq \max\{K, L\}$. With standard AC^0 operations on fields in $K \times L$-bit vectors plus* PackSignBits *and* Shuffle *and standard AC^0 operations on integers with M bits, we can simulate standard AC^0 operations on full words with $W = KL$ bits in constant time.*

Proof. We simply need to go through each of our word computations $\bar{\wedge}$, +, $\ll$, and msb. The easiest is $\bar{\wedge}$ since it doesn't matter that a word is divided into fields. To compute $\mathsf{msb}(X)$, we first set $x = (\texttt{PackSignBits}(\texttt{Mask}(X)))$ with x an M-bit integer x (using the C-convention that overflow is discarded). Next $i = \mathsf{msb}(x)$ gives us the field in X with the most significant bit. Now, $y = \mathsf{msb}(\texttt{Shuffle}(X, i))$ gives us this field in an integer y (using the C-convention that the integer i is padded with 0s to give a vector). Our final result is $(i \ll \ell) + \mathsf{msb}(y)$.

Addition is the most interesting case. We want to compute the full word addition $X + Y$. First, we compute the carries for each local field addition. An integer c viewed as a vector of carry bits is computed as

$$c = \texttt{PackSignBit}(\texttt{VecLess}(\texttt{VecADD}(X, Y), X))$$

Next, we compute an integer d of bits telling which fields get all 1s by the addition, hence for which the carry carries over to the next field.

$$d = \texttt{PackSignBit}(\texttt{VecEQ}(0, \texttt{VecNEG}(\texttt{VecADD}(X, Y))))$$

Now, the true carry bits for each field are collected in the integer

$$e = ((c \ll 1) + d) \oplus d$$

The final result is then

$$Z = \texttt{VecADD}(\texttt{VecADD}(X, Y), \texttt{UnPackLeastBit}(e))$$

defining $E = \texttt{UnPackLeastBit}(e)$ such that $E\langle i\rangle = e[i]$. To compute UnPackLeastBit, we want to shuffle the bits of e to the fields. First we cast e to a vector F. Then, if $i = a_i L + b_i$, $b_i < L$, we get $e[i] = F\langle a_i\rangle[b_i]$. Once and for all we construct universal vectors α and β

such that $\alpha\langle i\rangle = a_i = i \gg \ell$ and $\beta\langle i\rangle = 1 \ll b_i = 1 \ll (i - (a_i \ll \ell))$. Then

$$\texttt{UnPackLeastBit}(e) = \texttt{AND}(\texttt{Mask}(\texttt{AND}(\texttt{Shuffle}(F, \alpha), \beta)), \texttt{Shuffle}(1, 0)))$$

Here $\texttt{Shuffle}(1,0)$ is just a constant having 1 in each field.

The shifts are straightforward using `Shuffle` to do the global shifting of fields, and combining this with local shifting.

We note that even if our vector processor supports multiplication on fields, then this does not generalize to constant time multiplication on full words.

Theorem 3.1 immediately implies that we can use multimedia processors as standard AC^0 processors over the full word length. For example, Thorup has shown that this gives us sorting linear space sorting in $O(n \log\log n)$ expected time [34].

THEOREM 3.2. *If $K = W^{\Omega(1)}$, the processor from Theorem 3.1 supports fusion trees and atomic heaps.*

Proof. As shown by Andersson et al. [9], we can implement fusion trees and atomic heaps with AC^0 operations if we augment our standard AC^0 operations with the following operations:

- $\texttt{Duplicate}(x)$: Returns a word where each field is x.
- $Y = \texttt{Select}(X, S)$: The fields in S are viewed as bit pointers; a field is returned, containing the selected bits in X. That is, if $i < K$, $Y[i] = X[S\langle i\rangle]$.

  We do not need to implement `Select` directly. It suffices to maintain a representation of S such that $S\langle i\rangle$ can be set in constant time, and such that $Y = \texttt{Select}(X, S)$ can be computed in constant time.

First we note that `Duplicate` is trivially implemented as $\texttt{Shuffle}(x, 0)$.

To implement $Y = \texttt{Select}(X, S)$, we exploit that we only need an implicit representation of S. We will represent S via two vectors T and U. The correspondence is made as follows. In order to set $S\langle r\rangle = s$, we first set $i = s \gg \ell$ $(= s/L)$ and $j = s - (i \ll \ell)$, that is, $s = iL + j$. Then we set $T\langle r\rangle = i$ and $U\langle r\rangle = 1 \ll j$. Now,

$$\texttt{Select}(X, S) = \texttt{PackSignBits}(\texttt{Mask}(\texttt{AND}(\texttt{Shuffle}(X, T), U))$$

The reality of multimedia As warned earlier, the instruction sets on a real multimedia processors like the Pentium 4 appear rather ad-hoc from our perspective. Some vector instructions are only defined for some field-lengths, and there is no single field-length that can do it all. In particular, the `Shuffle` [25, p. 3-605–611] uses 16-bit fields, whereas `PackSignBits` [25, p. 3-576] uses 8-bit fields. However, we can use other packing instructions to pack 16-bit fields into 8-bit fields [25, pp. 3-522–526] and then apply the packing of sign-bits, the overall result being packing of 16-bit fields. Another dissatisfying aspect is that the `Shuffle` only works on 64 bits at the time, rather than 128 bits. All this seems pretty discouraging, but interestingly Motorola's AltiVec [30, p. 6-112] does offer a full 16×8-bit field `Shuffle`, complementing the Pentium's `PackSignBits`, perfectly. In fact, the AltiVec `Shuffle` is so powerful that it can pick out 16 arbitrary 8-bit fields from two 16×8-bit vectors. Unfortunately, the AltiVec does not have a `PackSignBits`.

However, the interesting lesson is that 16×8-bit `PackSignBits` and `Shuffle` are both implemented on current multimedia processors, with Pentium 4 being the closest to getting them both right simultaneously. For both operations, increasing the field length is easy in hardware as it just amounts to adding more parallel wires. Hence it should not be a technical problem to make a 16×64-bit processor, that we can use as a 1024-bit processor that can implement the Practical RAM, fusion trees, and atomic heaps.

We note that such a large word-length does change quite a lot from a practical perspective. For example, radix sort works great in practice for 32-bit integers, but for 1024-bit integers it will be hopelessly slow. That is, we really start feeling the gap between $\log n$ and W.

Acknowledgment I would like to thank Ken Ross and Nathan Slingerland for discussions about modern processors, and to Ken Ross in particular for pointing me to Intel's Pentium 4 [25].

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Princiles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[2] M. Ajtai, M. L. Fredman, and J. Komlós. Hash functions for priority queues. *Inform. and Comput.*, 63:217–225, 1984. Announced at FOCS'83.

[3] S. Albers and T. Hagerup. Improved parallel integer sorting without concurrent writing. *Inf. Contr.*, 136:25–51, 1997.

[4] S. Alstrup, D. Harel, P.W. Lauridsen, and M. Thorup. Dominators in linear time. *SIAM J. Comp.*, 28(6):2117–2132, 1999.

[5] S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. In *Proc. 39th FOCS*, pages 534–544, 1998.

[6] A. Andersson. Faster deterministic sorting and searching in linear space. In *Proc. 37th FOCS*, pages 135–141, 1996.

[7] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? *J. Comp. Syst. Sc.*, 57:74–93, 1998. See also STOC'95.

[8] A. Andersson, P.B. Miltersen, S. Riis, and M. Thorup. Static dictionaries on AC^0 RAMs: Query time $\Theta(\sqrt{\log n / \log\log n})$ is necessary and sufficient. In *Proc. 37th FOCS*, pages 441–450, 1996.

[9] A. Andersson, P.B. Miltersen, and M. Thorup. Fusion trees can be implemented with AC^0 instructions only. *Theor. Comp. Sc.*, 215(1-2):337–344, 1999.

[10] A. Andersson and M. Thorup. A pragmatic implemention of monotone priority queues. DIMACS Implementation challenge, 1996.

[11] A. Andersson and M. Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proc. 32th STOC*, 2000.

[12] P. Beame and F. Fich. Optimal bounds for the predecessor problem. In *Proc. 31st STOC*, pages 295–304, 1999.

[13] P. Beame and J. Håstad. Optimal bounds for decision problems on the CRCW PRAM. *J. ACM*, 36(3):643–670, 1989.

[14] A. M. Ben-Amram. When can we sort in $o(n \log n)$ time. *J. Comput. System Sci.*, 54:345–370, 1997. See also FOCS'93.

[15] A. Brodnik, S. Carlsson, J. Karlsson, and I. Munro. Worst case constant time priority queues. In *Proc. 12th SODA*, pages 523–528, 2001.

[16] A. Brodnik, P. B. Miltersen, and I. Munro. Trans-dichotomous algorithms without multiplication - some upper and lower bounds. In *Proc. 5th WADS, LNCS 1272*, pages 426–439, 1997.

[17] B. Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *J. ACM*, 47(6):1028–1047, 2000.

[18] P.F. Dietz. Optimal algorithms for list indexing and subset rank. In *Proc. WADS '89, LNCS 382*, pages 39–46, 1989.

[19] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, 1984.

[20] M. L. Fredman and M.E. Saks. The cell probe complexity of dynamic data structure. In *Proc. 21st STOC*, pages 345–354, 1989.

[21] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47:424–436, 1993. See also STOC'90.

[22] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci.*, 48:533–551, 1994. See also FOCS'90.

[23] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *J. Comput. System Sci.*, 30:209–221, 1985. Announced at STOC'83.

[24] IEEE. IEEE standard for binary floating-point arithmetic. *SIGPLAN Notices*, 22:9–25, 1985.

[25] Intel. The IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference (Order Number 245471), 2001. `http://developer.intel.com/design/pentium4/manuals/245471.htm`.

[26] B.W. Kernighan and D.M. Ritchie. *The C Programming Language (2nd ed.)*. Prentice Hall, 1988.

[27] E.M. McCreight. Priority search trees. *SIAM J. Comp.*, 14(2):257–276, 1985.

[28] K. Mehlhorn and S. Nähler. Bounded ordered dictionaries in $O(\log\log n)$ time and $O(n)$ space. *Inf. Proc. Lett.*, 35(4):183–189, 1990.

[29] P. B. Miltersen. Lower bounds for static dictionaries on RAMs with bit operations. In *Proc. 23rd ICALP, LNCS 1099*, pages 442–453, 1996.

[30] Motorola. AlitVec Technology Programming Evironments Manual, 2002. `http://e-www.motorola.com/brdata/PDFDB/docs/ALTIVECPEM.pdf`.

[31] B. Stroustrup. *The C++ Programming Language, Special Edition.* Addison-Wesley, Reading, MA, 2000.

[32] M. Thorup. Faster deterministic sorting and priority queues in linear space. In *Proc. 9th SODA*, pages 550–555, 1998.

[33] M. Thorup. Undirected single source shortest paths with positive integer weights in linear time. *J. ACM*, 46(3):362–394, 1999. See also FOCS'97.

[34] M. Thorup. Randomized sorting in $O(n \log\log n)$ time and linear space using addition, shift, and bit-wise boolean operations. *J. Algor.*, 42(2):205–230, 2002. Announced at SODA'97.

[35] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Syst. Theory*, 10:99–127, 1977.

[36] D. E. Willard. Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion tree. *SIAM J. Comput.*, 29(3):1030–1049, 2000. See also SODA'92.